

LOG8415
Advanced Concepts of Cloud Computing
Scaling Databases and Implementing Cloud Design Patterns

Vahid Majdinasab
Département Génie Informatique et Génie Logiciel
École Polytechnique de Montréal, Québec, Canada
`vahid.majdinasab[at]polymtl.ca`

Ali Hazime (2270891)

GitHub: <https://github.com/AliSandro104/LOG8415-final-project>

December 28, 2023

1 Benchmarking MySQL stand-alone vs MySQL Cluster

Execution steps:

1. On your local machine, run the bash script *create_instances.sh* to create and configure all ec2 instances needed
2. After SSH'ing to each instance, run the following scripts to automate the configuration:
 - If the instance is the stand-alone node, run the bash script *install_mysql.sh*
 - If the instance is the manager node, run the bash script *setup_manager.sh*
 - If the instance is the worker node, run the bash script *setup_worker.sh*
3. Now, on the stand-alone node, execute the following commands:
 - `sudo mysql -u root -p`
 - Press enter if prompted for a password (default password)
 - After connecting to the MySQL server, run the following SQL queries to install the Sakila database
 - (a) `SOURCE /tmp/sakila-db/sakila-schema.sql;`
 - (b) `SOURCE /tmp/sakila-db/sakila-data.sql;`
 - (c) `USE sakila;`
 - (d) `quit`
 - Execute the following commands for the *sysbench* benchmarking:
 - (a) `sudo sysbench --db-driver=mysql --mysql-user='root' --mysql-password="" --mysql-db=sakila --table-size=1000000 --tables=10 --time=60 --threads=6 oltp_read_write prepare`
 - (b) `sudo sysbench --db-driver=mysql --mysql-user='root' --mysql-password="" --mysql-db=sakila --table-size=1000000 --tables=10 --time=60 --threads=6 oltp_read_write run`
4. Observe the benchmarking results obtained, which are printed to stdout
5. Now, execute the following commands on the manager node to setup the cluster:
 - `source /etc/profile.d/mysqlc.sh`
 - `sudo /opt/mysqlcluster/home/mysqlc/bin/ndb_mgmd -f /opt/mysqlcluster/deploy/conf/config.ini --initial --configdir=/opt/mysqlcluster/deploy/conf`
 - Go to each worker instance and execute the following commands:
 - (a) `source /etc/profile.d/mysqlc.sh`
 - (b) `ndbd -c ip-XX-XX-XX-XXX.ec2.internal:1186`
, where ip-XX-XX-XX-XXX.ec2.internal represents the full private IP address of the manager node
 - Go back to the manager node
 - Before proceeding, check the status of the cluster using the command: `ndb_mgm -e show`

- `sudo env "PATH=$PATH" mysqld --defaults-file=/opt/mysqlcluster/deploy/conf/my.cnf --user=root &`
- `mysql -h 127.0.0.1 -u root -p`
- Press enter if prompted for a password (default password)
- After connecting to the MySQL server, run the following SQL queries to install the Sakila database
 - (a) `SOURCE initialize_db.sql`
 - (b) `quit`
- Execute the following commands for the *sysbench* benchmarking:
 - (a) `sysbench --db-driver=mysql --mysql-host=127.0.0.1 --mysql-user='root' --mysql-password="" --mysql-db=sakila --table.size=1000000 --tables=10 --time=60 --threads=6 oltp_read_write prepare`
 - (b) `sysbench --db-driver=mysql --mysql-host=127.0.0.1 --mysql-user='root' --mysql-password="" --mysql-db=sakila --table.size=1000000 --tables=10 --time=60 --threads=6 oltp_read_write run`
- Observe the benchmarking results obtained for the cluster, which are printed to stdout, and compare them to the results for the stand-alone node

Summary of Results:

- The configuration used for the benchmarking is the following and remained constant for both the standalone node and the cluster:
 1. Table size = 1000000
 2. Number of tables = 10
 3. Time = 60 seconds
 4. Number of threads = 6

For the standalone node, a total of 120,340 queries were performed, among which were 84,238 read queries, 24,068 write queries and 12,034 other queries. These represent 6,017 transactions, which means that we have a rate of 100.23 transactions completed per second. The minimum and maximum latency are 13.41 ms and 291.99 ms respectively, while the average latency is 59.84 ms. For the cluster, a total of 246,240 queries were performed, among which were 172,368 read queries, 49,248 write queries and 24,624 other queries. These represent 12,312 transactions, which means that we have a rate of 205.10 transactions completed per second. The minimum and maximum latency are 3.06 ms and 561.14 ms respectively, while the average latency is 29.24 ms.

Clearly, the MySQL cluster is performing better on average, as it is completing transactions twice as fast as the standalone node. This is the expected behaviour, as we have four different instances processing requests, as opposed to the standalone node. However, it must be noted that the maximum latency for the cluster is curiously much higher than that of the standalone node, indicating a higher variance in the response time of the cluster. This means that there exist cases where the standalone node actually performs better.

```
ubuntu@ip-172-31-44-226:~$ sudo sysbench --db-driver=mysql --mysql-user='root'
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)
```

Running the test with following options:

Number of threads: 6

Initializing random number generator from current time

Initializing worker threads...

Threads started!

SQL statistics:

queries performed:

read: 84238

write: 24068

other: 12034

total: 120340

transactions: 6017 (100.23 per sec.)

queries: 120340 (2004.62 per sec.)

ignored errors: 0 (0.00 per sec.)

reconnects: 0 (0.00 per sec.)

General statistics:

total time: 60.0290s

total number of events: 6017

Latency (ms):

min: 13.41

avg: 59.84

max: 291.99

95th percentile: 104.84

sum: 360067.22

Threads fairness:

events (avg/stddev): 1002.8333/4.37

execution time (avg/stddev): 60.0112/0.01

```
ubuntu@ip-172-31-44-226:~$
```

Figure 1: Benchmarking Results for the Standalone Node

```

ubuntu@ip-172-31-39-73:~$ sysbench --db-driver=mysql --mysql-host=127.0.0.1 --mysql
sysbench 1.0.20 (using system LuaJIT 2.1.0-beta3)

Running the test with following options:
Number of threads: 6
Initializing random number generator from current time

Initializing worker threads...

Threads started!

SQL statistics:
  queries performed:
    read:                172368
    write:               49248
    other:               24624
    total:              246240
  transactions:         12312 (205.10 per sec.)
  queries:              246240 (4102.03 per sec.)
  ignored errors:        0      (0.00 per sec.)
  reconnects:            0      (0.00 per sec.)

General statistics:
  total time:            60.0268s
  total number of events: 12312

Latency (ms):
  min:                   3.06
  avg:                   29.24
  max:                   561.14
  95th percentile:      64.47
  sum:                   360033.65

Threads fairness:
  events (avg/stddev):   2052.0000/22.33
  execution time (avg/stddev): 60.0056/0.00

ubuntu@ip-172-31-39-73:~$ █

```

Figure 2: Benchmarking Results for the Cluster

2 Implementation of the Proxy pattern

Execution steps: The bash script *create_instances.sh* that was executed previously creates the proxy instance and transfers the python script *proxy.py*. The only step here is to run that python script on the proxy instance, which will start listening for requests from the trusted host. Similarly, run the python script *worker.py* on each worker instance and the python script *master.py* on the manager node, to listen for requests coming from the proxy.

Pattern discussion: When creating the proxy instance, we must protect it by assigning it to a security group that has specific inbound and outbound rules. Indeed, this restricts incoming and outgoing traffic to the absolute minimum, preventing unauthorized access.

As we can see in the figure below, we permit SSH connections to the proxy by opening port 22. Additionally, we allow both-way communication on port 8082 with instances that are part of the MySQL cluster, and on port 8081 with the trusted host instance. Furthermore, we enable ping of the MySQL cluster using the 'ICMP' protocol.

```
226
227 # add ip permissions for proxy security group
228 add_inbound_ip_permissions(proxy_sg, 'tcp', 22, 22, ip_ranges, None)
229 add_inbound_ip_permissions(proxy_sg, 'tcp', 8082, 8082, ip_ranges, cluster_sg)
230 add_inbound_ip_permissions(proxy_sg, 'tcp', 8081, 8081, ip_ranges, trusted_host_sg)
231 add_inbound_ip_permissions(proxy_sg, 'icmp', -1, -1, ip_ranges, cluster_sg) # allow the proxy to receive a response after pinging the cluster
232 add_outbound_ip_permissions(proxy_sg, 'tcp', 8082, 8082, ip_ranges, cluster_sg)
233 add_outbound_ip_permissions(proxy_sg, 'tcp', 8081, 8081, ip_ranges, trusted_host_sg)
234 add_outbound_ip_permissions(proxy_sg, 'icmp', -1, -1, ip_ranges, cluster_sg) # allow the proxy to ping the cluster
235
```

Figure 3: Inbound and Outbound Rules for the Proxy Security Group

Now that the instance is secured, we need to code the functionality of the proxy using socket programming in python. The proxy will start listening on port 8081 for requests that can only arrive from the trusted host, due to the security group restriction. Once it receives the data, the proxy has to select the node from the MySQL cluster that will perform the operation before forwarding the query.

```
66 proxy_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
67 proxy_socket.bind((proxy_host, proxy_port))
68 proxy_socket.listen(1)
69 print(f"Proxy listening on {proxy_host}:{proxy_port}")
70
71 # listen on port 8081 for requests from the trusted host
72 while True:
73     conn, addr = proxy_socket.accept()
74     print(f"Connection from {addr}")
75
76     data = conn.recv(1024).decode('utf-8')
77     print(f"Received data: {data}")
78
79     node_ip_address = choose_node(data)
80     node_port = 8082
```

Figure 4: Part of the Proxy Main Method

The data received contains the SQL query and the node selection algorithm to use. If the query

requires a write transaction, it is immediately transferred to the master node, as required by the instructions. Otherwise, if we have a read operation and the algorithm to use is 'Random', the proxy simply chooses a random worker from the list.

```

32 def choose_node(data):
33     # Open the config file to get the private ip of the cluster nodes
34     filename = 'cluster_private_ip.txt'
35     with open(filename, 'r') as file:
36         ip_addresses = file.readlines()
37
38     ip_addresses = [ip.strip() for ip in ip_addresses]
39
40     sql_query, proxy_algorithm = data.split('|')
41
42     sql_query_type = sql_query.split()[0].lower() # get the first word of the sql query to know if it's a read or write transaction
43
44     algorithm = worker_choice_algorithms[int(proxy_algorithm) - 1] # Decode the algorithm choice for the worker selection
45
46     if algorithm == 'Direct hit' or sql_query_type != "select" : # if the algorithm is 'Direct hit' or if the sql query requires a write transaction
47         return ip_addresses[1] # return the ip of the master node
48
49     elif algorithm == 'Random': # if the algorithm is 'Random'
50         random_list = ip_addresses[-3:] # take the last three elements of the list (representing the ip addresses of the worker nodes)
51         return random.choice(random_list) # return the ip address of a random worker
52
53     elif algorithm == 'Customized': # if the algorithm is 'Customized'
54         return customized_algorithm(ip_addresses[-3:]) # call method that will compute the response time of each worker and choose the smallest one
55

```

Figure 5: Proxy Node Selection Algorithm

Finally, if the query requires a read transaction and the algorithm to use is 'Customized', the proxy will ping each worker node once to determine the node with the lowest response time. The code used was inspired by this [article](#).

```

9 # measure the response time when ping a server
10 # code inspired by https://medium.com/@networksautomation/python-ping-an-ip-address-663ed902e051
11 def measure_response_time(ip_address):
12     data = ""
13     output = Popen(f"ping {ip_address} -c 1", stdout=PIPE, encoding="utf-8", shell=True)
14
15     for line in output.stdout:
16         data = data + line
17         time_match = findall(r"time=(\d+\.\d+)", data) # search for the round-trip time in the data string
18
19     if time_match:
20         return float(time_match[-1]) # if a time was found in the output, we return it
21     else:
22         return float('inf') # else, return infinity
23
24 def customized_algorithm(ip_addresses_workers):
25     response_times = {}
26
27     for ip_address in ip_addresses_workers:
28         response_times[ip_address] = measure_response_time(ip_address) # measure the response time of each worker
29
30     return min(response_times, key=response_times.get) # get the key which is the ip address corresponding to the minimum response time

```

Figure 6: Determination of the Lowest Ping Response Time

After the proxy chooses a node and forwards the data, the selected node will receive it while listening on port 8082 for incoming requests from the proxy.

```

67 master_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
68 master_socket.bind((worker_host, worker_port))
69 master_socket.listen(1)
70 print(f"Worker {worker_num} listening on {worker_host}:{worker_port}")
71
72 # listen on port 8082 for requests from the proxy
73 while True:
74     conn, addr = master_socket.accept()
75     print(f"Connection from {addr}")
76
77     data = conn.recv(1024).decode('utf-8')
78     print(f"Received data: {data}")
79
80     # Process the SQL query
81     result = process_sql_query(data, worker_num)
82     result = f"{result}|Worker{worker_num}"
83
84     # Send the result back to the proxy
85     conn.sendall(str(result).encode('utf-8'))
86
87     conn.close()

```

Figure 7: Worker Receiving the Data from the Proxy

Then, the node will connect to the Sakila database hosted on the master node and execute the SQL query. The result is sent back to the proxy, with the name of the node that did the work. Both of those strings will eventually be displayed on the gatekeeper web application. It must be noted that the implementation of the manager and the worker is very similar, with the only exception being the fact that the master node uses localhost for the database configuration instead of the private ip address of the host.

```

26 # sql config
27 db_config = {
28     'host': ip_addresses[1],
29     'user': 'worker{}'.format(worker_num),
30     'password': 'worker{}'.format(worker_num),
31     'database': 'sakila',
32 }
33
34 try:
35     # Connect to the MySQL server
36     connection = mysql.connector.connect(**db_config)
37     cursor = connection.cursor()
38
39     # Execute the SQL query
40     cursor.execute(sql_query)
41
42     # Fetch the result
43     result = cursor.fetchall()
44
45     # Close the cursor and connection
46     cursor.close()
47     connection.close()
48
49     return result

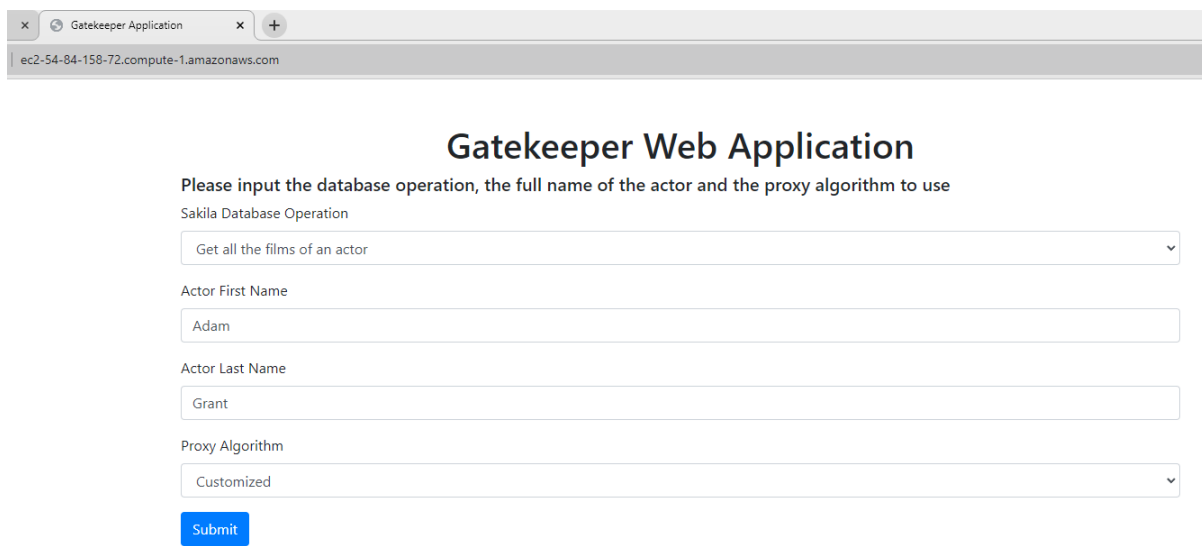
```

Figure 8: Worker Executing the SQL Query

3 Implementation of the Gatekeeper pattern

Execution steps: The bash script *create_instances.sh* that was executed previously creates the gatekeeper and the trusted host instances. From the local machine, the gatekeeper receives the bash script *deploy_flask_app.sh* and two html templates *index.html* and *result.html*, while the trusted host receives the python script *trusted_host.py*.

1. Run the python script *trusted_host.py* on the trusted host instance, which will start listening for requests from the gatekeeper application
2. Run the script *deploy_flask_app.sh* on the gatekeeper instance to deploy the flask application
3. Navigate in your browser to the public IP address of the gatekeeper instance to launch the web application
4. Use the application to send requests to the trusted host



The screenshot shows a web browser window with the title "Gatekeeper Application" and the address bar displaying "ec2-54-84-158-72.compute-1.amazonaws.com". The main content area has the heading "Gatekeeper Web Application" and a subheading "Please input the database operation, the full name of the actor and the proxy algorithm to use". Below this, there are four input fields: "Sakila Database Operation" (a dropdown menu with "Get all the films of an actor" selected), "Actor First Name" (a text input field with "Adam" entered), "Actor Last Name" (a text input field with "Grant" entered), and "Proxy Algorithm" (a dropdown menu with "Customized" selected). At the bottom left of the form is a blue "Submit" button.

Figure 9: Gatekeeper Web Application - Home Page

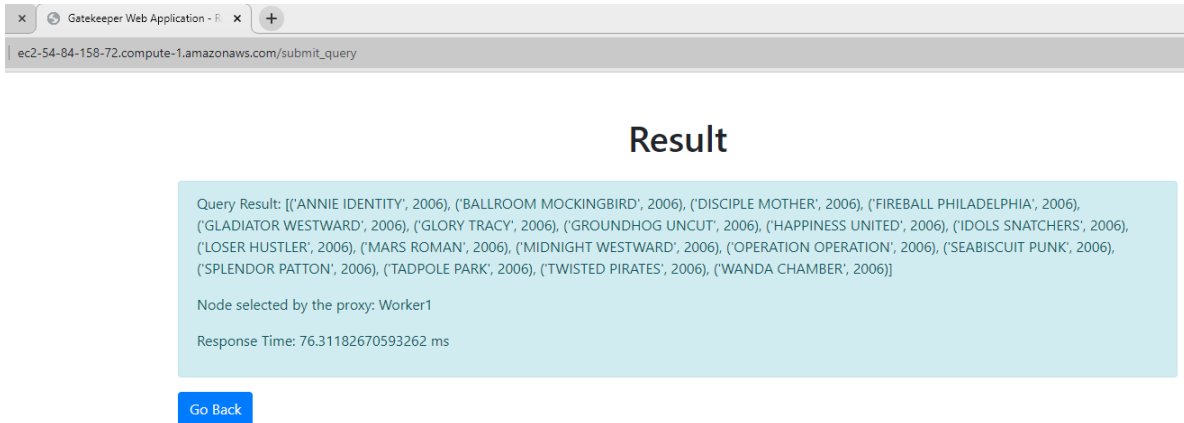


Figure 10: Gatekeeper Web Application - Query Result Page

Pattern discussion: When creating the gatekeeper and the trusted host instances, we must protect them with security groups that have specific inbound and outbound rules. Indeed, this restricts incoming and outgoing traffic to the absolute minimum, preventing unauthorized access.

As we can see in the figure below, we permit SSH connections by opening port 22. Additionally, we allow both-way communication on port 8080 between the gatekeeper and the trusted host. Furthermore, we enable communication between the trusted host and the proxy on port 8081, while also permitting HTTP (port 80) and HTTPS (port 443) connections on the gatekeeper to receive requests from the web application. All other traffic is forbidden for these security groups.

```

236 # add ip permissions for trusted host security group
237 add_inbound_ip_permissions(trusted_host_sg, 'tcp', 22, 22, ip_ranges, None)
238 add_inbound_ip_permissions(trusted_host_sg, 'tcp', 8081, 8081, ip_ranges, proxy_sg)
239 add_inbound_ip_permissions(trusted_host_sg, 'tcp', 8080, 8080, ip_ranges, gatekeeper_sg)
240 add_outbound_ip_permissions(trusted_host_sg, 'tcp', 8081, 8081, ip_ranges, proxy_sg)
241 add_outbound_ip_permissions(trusted_host_sg, 'tcp', 8080, 8080, ip_ranges, gatekeeper_sg)
242
243 # add ip permissions for gatekeeper security group
244 add_inbound_ip_permissions(gatekeeper_sg, 'tcp', 22, 22, ip_ranges, None)
245 add_inbound_ip_permissions(gatekeeper_sg, 'tcp', 80, 80, ip_ranges, None)
246 add_inbound_ip_permissions(gatekeeper_sg, 'tcp', 443, 443, ip_ranges, None)
247 add_inbound_ip_permissions(gatekeeper_sg, 'tcp', 8080, 8080, ip_ranges, trusted_host_sg)
248 add_outbound_ip_permissions(gatekeeper_sg, 'tcp', 8080, 8080, ip_ranges, trusted_host_sg)

```

Figure 11: Inbound and Outbound Rules for the Gatekeeper and Trusted Host Security Group

Now, the web application on the gatekeeper is very basic. It takes as input from the user the database operation, the full name of the actor and the node selection algorithm to be used by the proxy. To prevent the user from exploiting the application with SQL injections, the database operations are limited to a drop-down menu, from which the user must select an option. The available operations are the following: Get the info of an actor, add an actor, delete an actor, and get all the films of an actor. Therefore, we have two read transactions and two write transactions.

```

10 <body>
11 <div class="container mt-5">
12 <h1 style="text-align: center;">Gatekeeper Web Application</h1>
13 <form action="/submit_query" method="post">
14 <h5>Please input the database operation, the full name of the actor and the proxy algorithm to use</h5>
15
16 <!-- Input field for the database operation-->
17 <div class="form-group">
18 <label for="operation">Sakila Database Operation</label>
19 <select class="form-control" name="operation" id="operation">
20 <option value="1">Get the basic info of an actor</option>
21 <option value="2">Add an actor</option>
22 <option value="3">Delete an actor</option>
23 <option value="4">Get all the films of an actor </option>
24 </select>
25 </div>

```

Figure 12: Database Operations Available to the User

After the user submits their input values, the gatekeeper receives them, and calls the method *send_data_to_trusted_host()*, as shown on line 78 in the code snippet below. This function concatenates those input values into one string, and sends the latter to the trusted host that is listening on port 8080.

```

65 @app.route('/submit_query', methods=['POST'])
66 def submit_query():
67
68     # Record the start time
69     start_time = time.time()
70
71     # Get the inputs from the user interface
72     operation = request.form['operation']
73     actor_first_name = request.form.get('first_name', '')
74     actor_last_name = request.form.get('last_name', '')
75     proxy_algorithm = request.form['proxy']
76
77     # Send the db operation, user input and proxy algorithm to the trusted host
78     result = send_data_to_trusted_host(operation, actor_first_name, actor_last_name, proxy_algorithm)
79
80     query_result, node_selected = result.split('|')
81
82     # Calculate the response time
83     response_time = time.time() - start_time
84
85     return render_template('result.html', result=query_result, node_selected=node_selected, response_time=response_time)

```

Figure 13: Data Received and Forwarded to the Trusted Host

Then, the response is processed upon arrival and the total response time is measured. Those two values are sent to the HTML template *result.html* that will display them to the user.

```

10 <body>
11   <div class="container mt-5">
12     <h1 style="text-align: center;" class="mb-4">Result</h1>
13     <div class="alert alert-info" role="alert">
14       <!-- Display the results -->
15       <p>Query Result: {{ result }}</p>
16       <p>Node selected by the proxy: {{ node_selected }}</p>
17
18       <!-- Display response time -->
19       <p>Response Time: {{ response_time * 1000}} ms</p>
20     </div>
21     <a class="btn btn-primary" href="/" role="button">Go Back</a>
22   </div>

```

Figure 14: Rendering the Results and the Total Response Time

Next, we discuss the trusted host design. When it receives data from the gatekeeper, it must perform some input validation. First, we check whether we receive four input values from the gatekeeper: the database operation, the first name of the actor, the last name of the actor, and the node selection algorithm that the proxy will use.

```

39 # listen on port 8080 for requests from the gatekeeper
40 while True:
41     conn, addr = trusted_host_socket.accept()
42     print(f"Connection from {addr}")
43
44     data = conn.recv(1024).decode('utf-8')
45     print(f"Received data: {data}")
46
47     # Split the received data into components
48     components = data.split('|')
49
50     # Validate the request
51     if len(components) == 4:
52         operation, first_name, last_name, proxy_algorithm = components
53
54         # Check if the inputs are valid
55         if validate_request(first_name, last_name):
56             # Construct the SQL query based on the operation and user input
57             sql_query = construct_sql_query(operation, first_name, last_name)
58
59             # add the proxy algorithm to the data to be sent
60             data = f"{sql_query}|{proxy_algorithm}"

```

Figure 15: Number of Inputs Validation

Then, we check whether the format of the actor's first name and last name is valid. Since we don't have specific requirements for this validation, it was decided that only non-empty alphabetic characters are allowed. This is mainly to prevent SQL injections that can look like 'OR 1=1' and '; DROP TABLE Actor'.

```

3  def validate_request(first_name, last_name):
4      # Check if both first name and last name are non-empty and contain only alphabetic characters
5      if first_name.isalpha() and last_name.isalpha():
6          return True
7      return False

```

Figure 16: Actor's First Name and Last Name Validation

Finally, once the validation is completed, we can safely build the SQL query based on the database operation, and the actor's first name and last name. The SQL query is sent to the proxy, along with the node selection algorithm for further manipulation, as explained in the proxy pattern section.

```

9  def construct_sql_query(operation, first_name, last_name):
10     # Construct the sql query based on the operation chosen by the user
11     if operation == "1":
12         return f"SELECT * FROM actor WHERE actor.first_name = '{first_name}' AND a
13     elif operation == "2":
14         return f"INSERT INTO actor (first_name, last_name) VALUES ('{first_name}',
15     elif operation == "3":
16         return f"DELETE FROM actor WHERE actor.first_name = '{first_name}' AND act
17     elif operation == "4":
18         return f"SELECT film.title, film.release_year FROM film JOIN film_actor ON
19     else:
20         return "Error: Invalid operation"

```

Figure 17: SQL Query Construction Based on Database Operation Selected

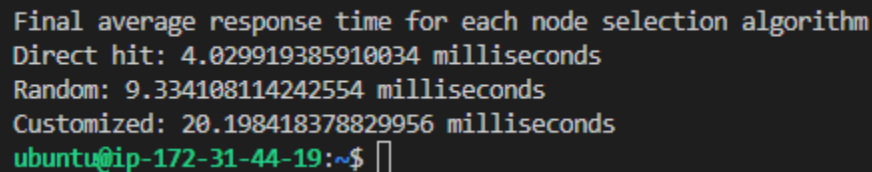
In summary, we notice that all instances in the design play a crucial role in the functioning of the application. And more importantly, they are secure because of the inbound and outbound rules defined for their respective security groups. For example, the trusted host can only communicate with the gatekeeper on port 8080, and with the proxy on port 8081. It is also possible to SSH to the trusted host, but this is only to help the developer configure it accordingly. Absolutely no other traffic is allowed for this instance.

4 Benchmarking the Cloud Patterns:

Execution steps: The bash script *create_instances.sh* that was executed previously creates the gatekeeper, the trusted host and the proxy instances. From the local machine, the gatekeeper receives the python script *benchmarking.py*. The only step here is to run that python script on the gatekeeper instance to send 1000 requests for each one of the three node selection algorithms used by the proxy. Then, we can compare the average response times obtained.

Note: Make sure that the script *trusted_host.py* is running on the trusted host instance, the script *proxy.py* is running on the proxy instance, the script *worker.py* is running on each worker instance and the script *master.py* is running on the manager node, as instructed in the previous sections.

Summary of results: As we can see in the figure below, the average response time is 4.03 ms for the 'Direct Hit' algorithm, 9.33 ms for the 'Random' algorithm, and 20.20 ms for the 'Customized' algorithm. Interestingly, the 'Customized' algorithm is the slowest despite being the one that chooses the optimal node. Indeed, pinging each worker and choosing the one that responds the fastest is time-consuming and not beneficial in this context, in which there is only one client sending one request at a time. On the other hand, the 'Direct Hit' algorithm is the fastest, most likely because the master node hosts the Sakila database and does not need to connect to another instance to access it, like the workers do in the 'Random' algorithm.



```
Final average response time for each node selection algorithm
Direct hit: 4.029919385910034 milliseconds
Random: 9.334108114242554 milliseconds
Customized: 20.198418378829956 milliseconds
ubuntu@ip-172-31-44-19:~$
```

Figure 18: Average Response Time for Each Node Selection Algorithm